# Contents

# Exercise 1- Unsupervised Learning and Image Denoising

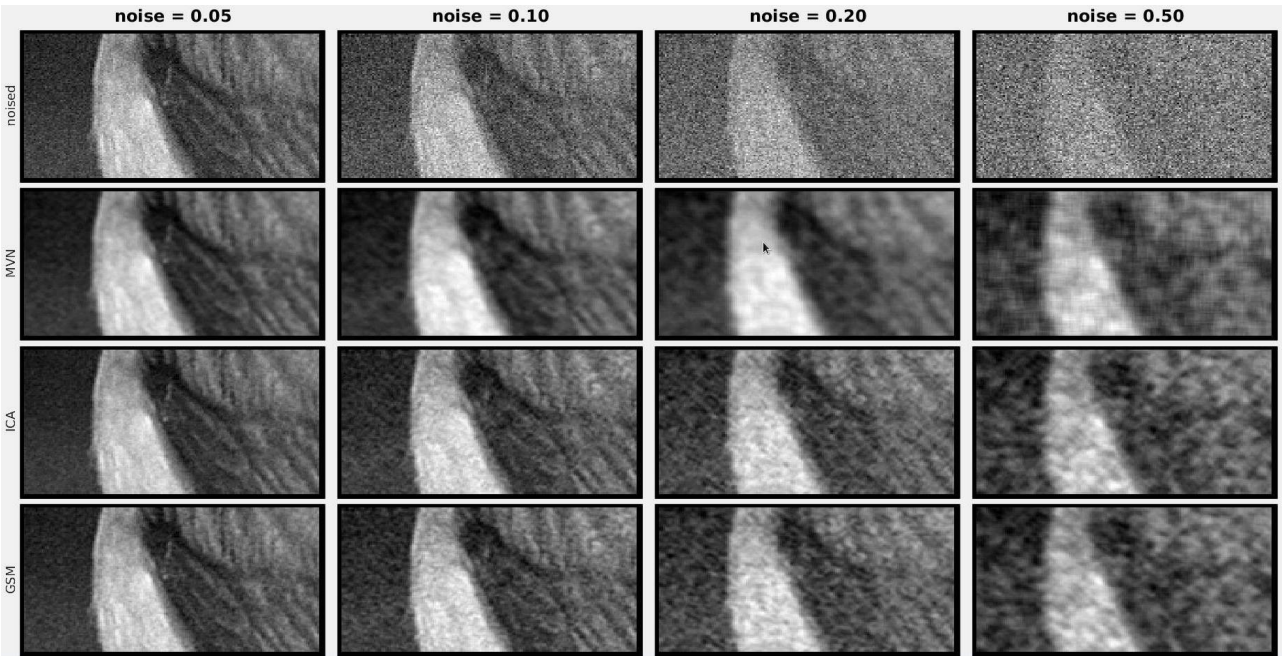
**Name:** Dan Kufra ID: 302190822 Username: dan_kufra
**Name:** Eran Malach ID: 200973915 Username: emalach
7 December 2016

## Introduction

In this exercise we implemented three different models for denoising images. We will provide a few comparisons between each model using different parameters and noises. In particular, we will compare the likelihood and pSNR of each model. We will also add our own visual comparison of which one "looks" better.

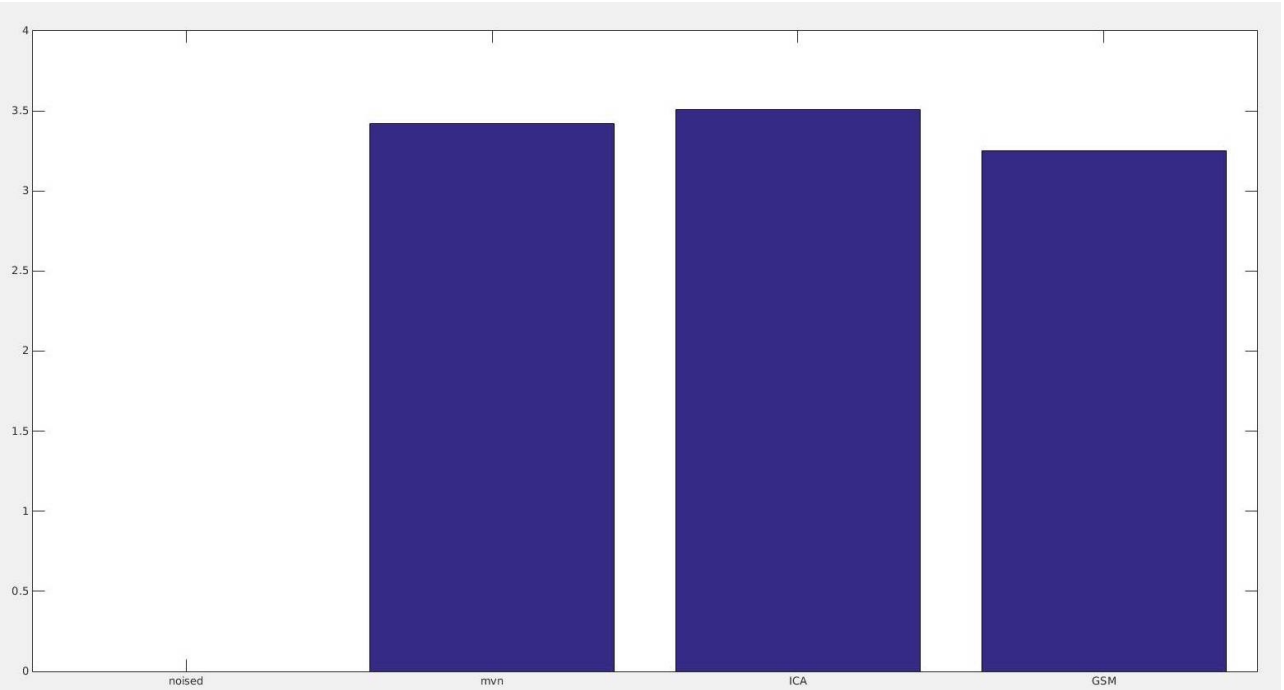
### Denoising patches in images:

### 5 gaussians:

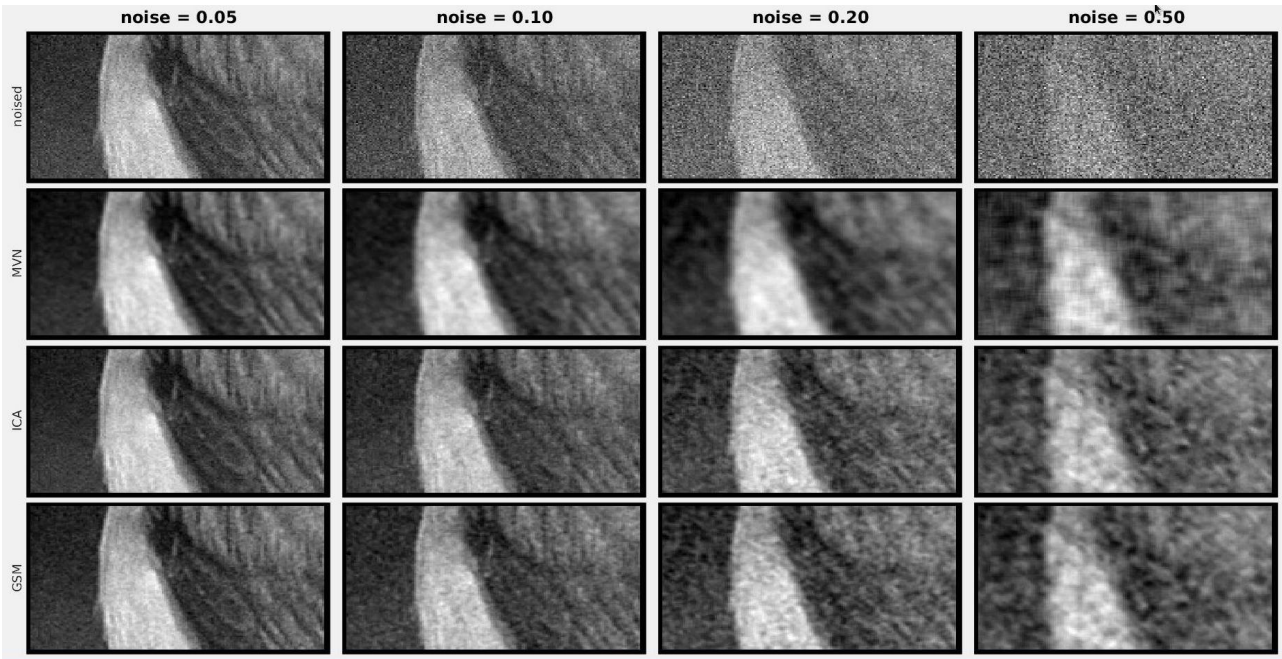


## pSNR:

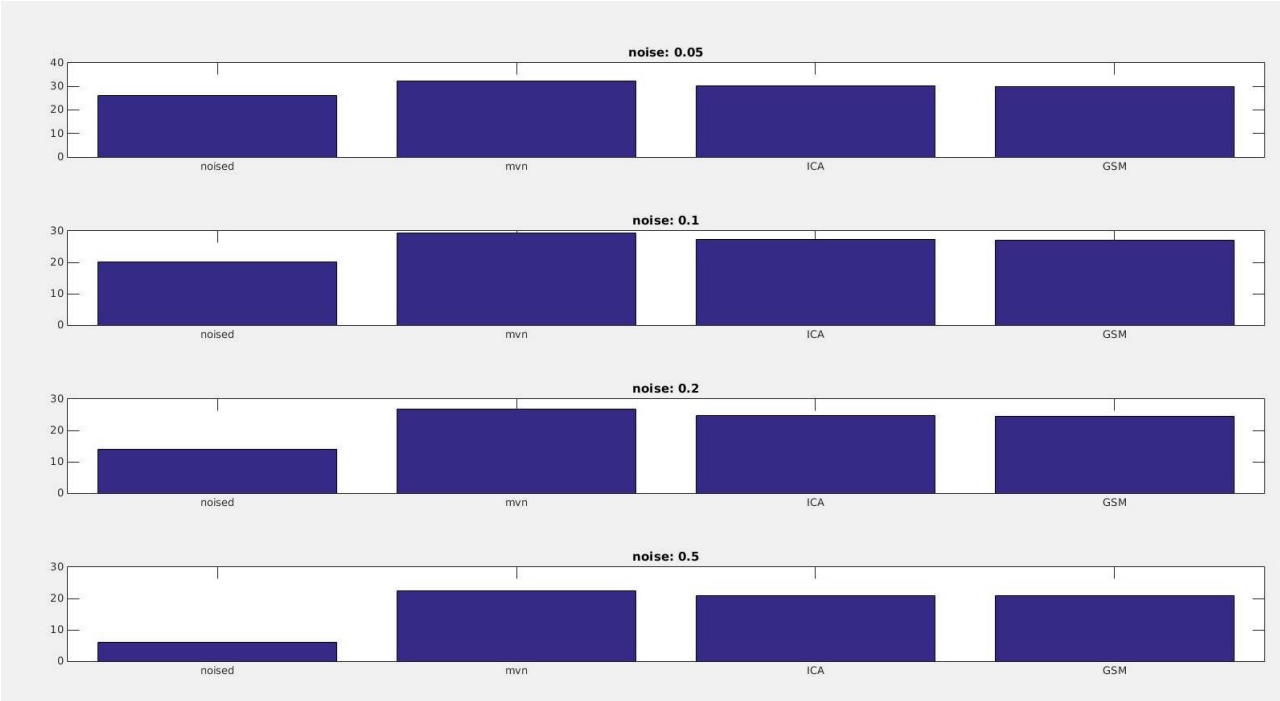

## Log Likelihood:



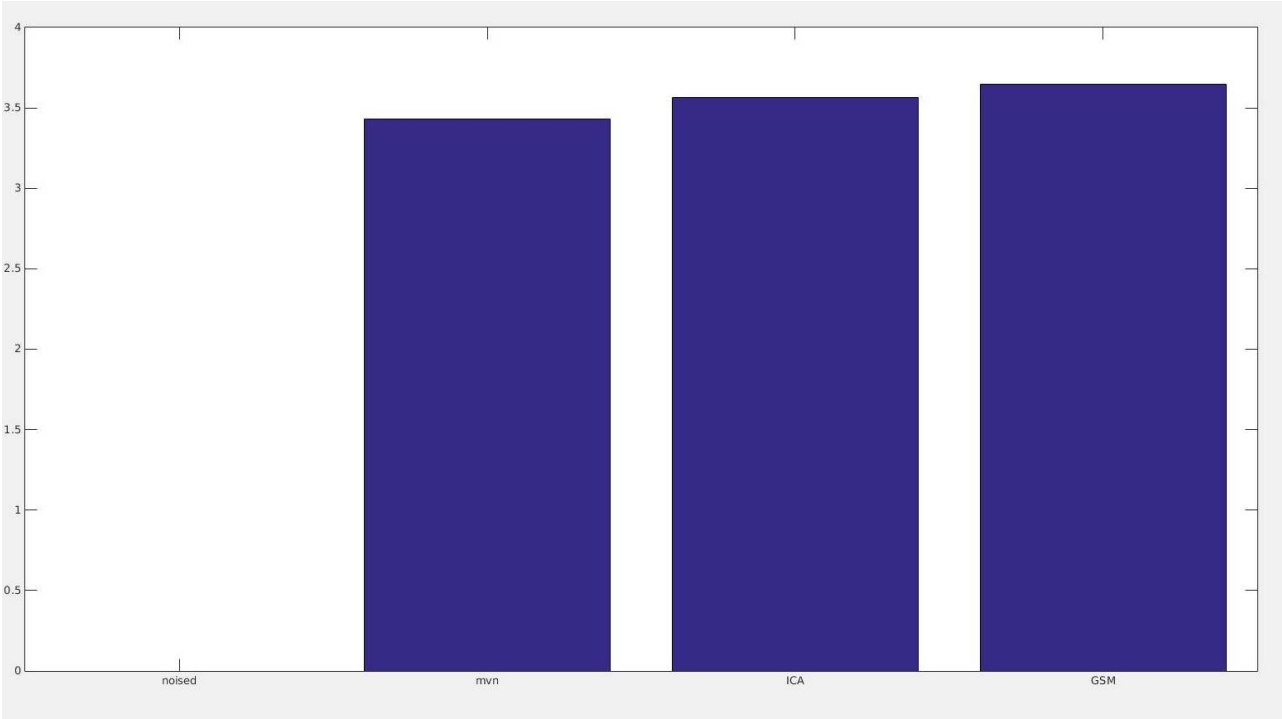
# 15 gaussians:



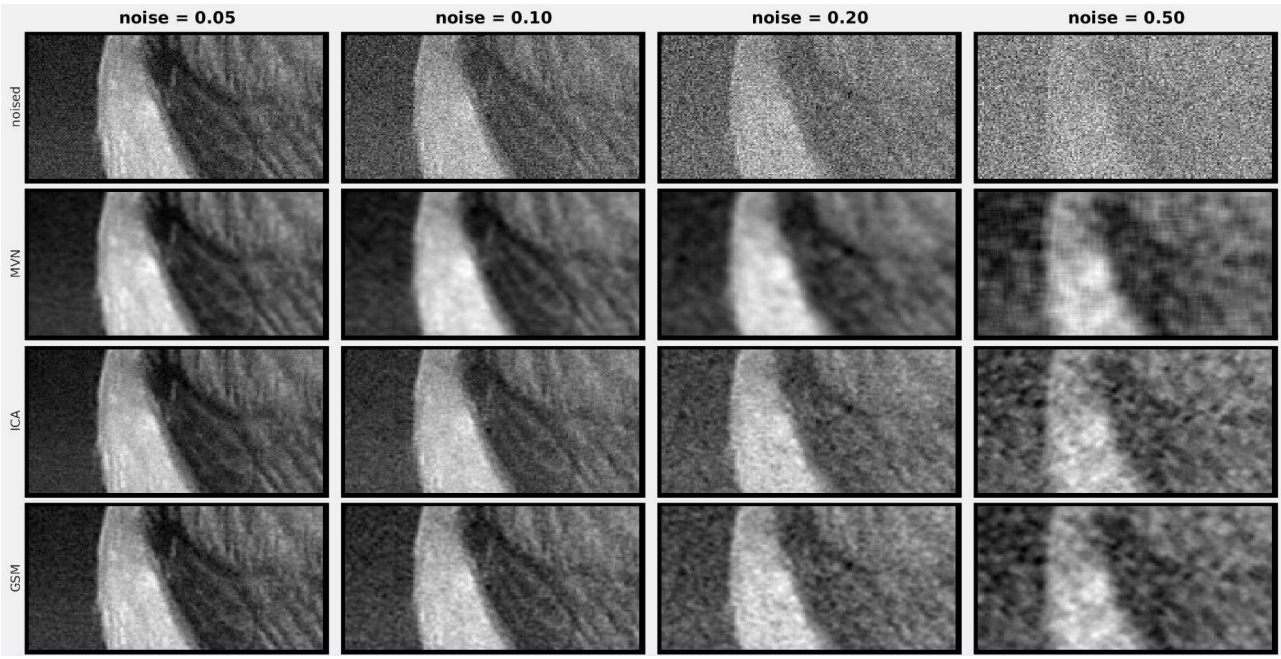

## pSNR:



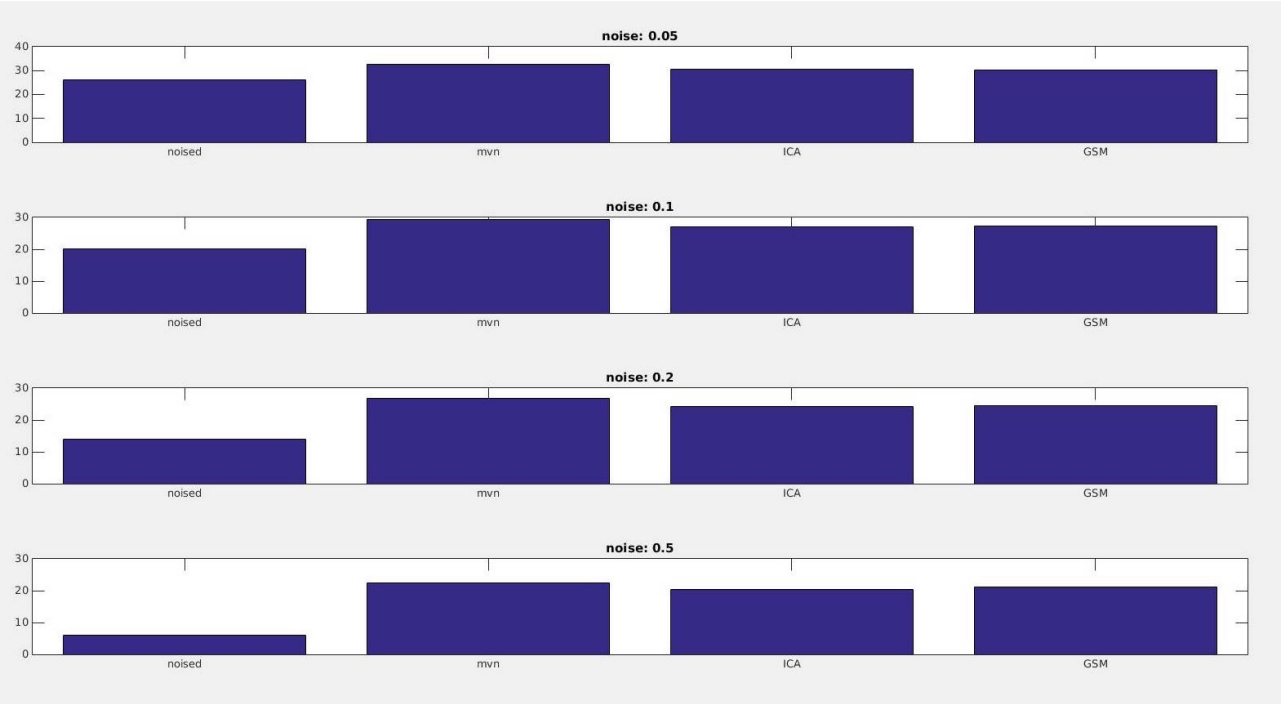
**Log Likelihood:**



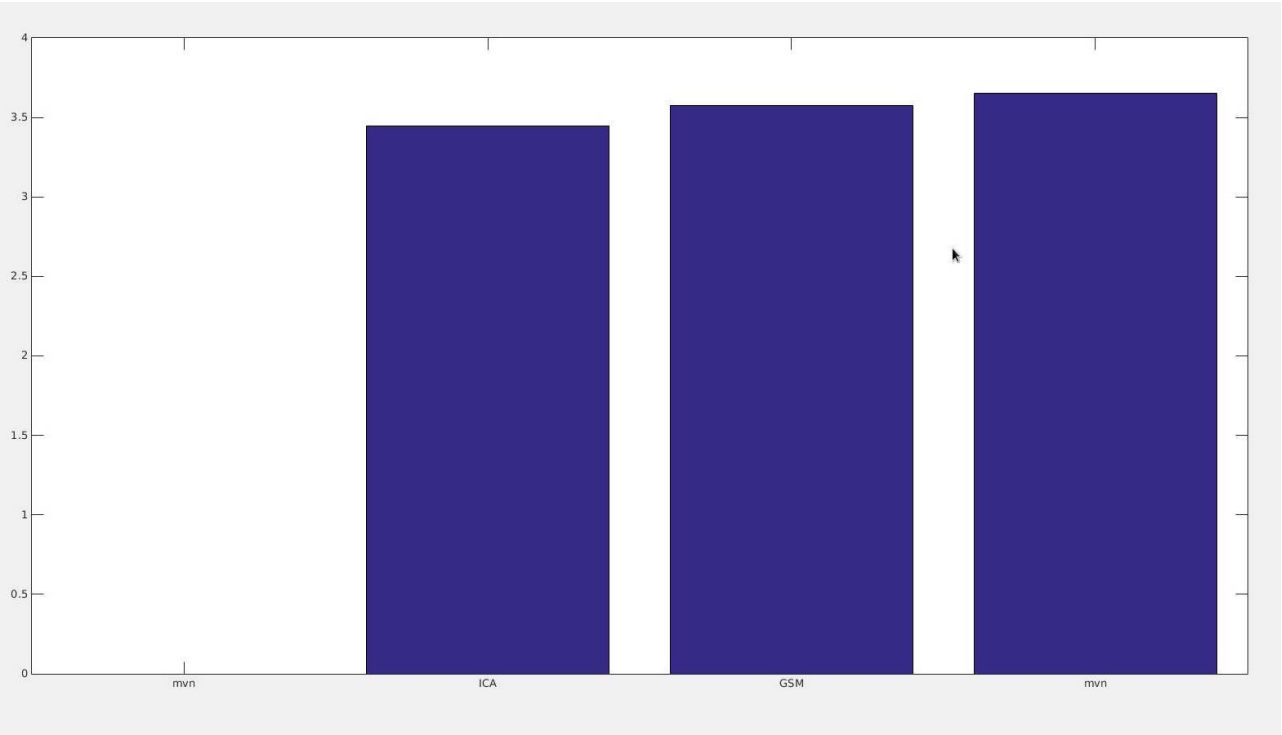

## 35 gaussians:



## pSNR:




## Log Likelihood:

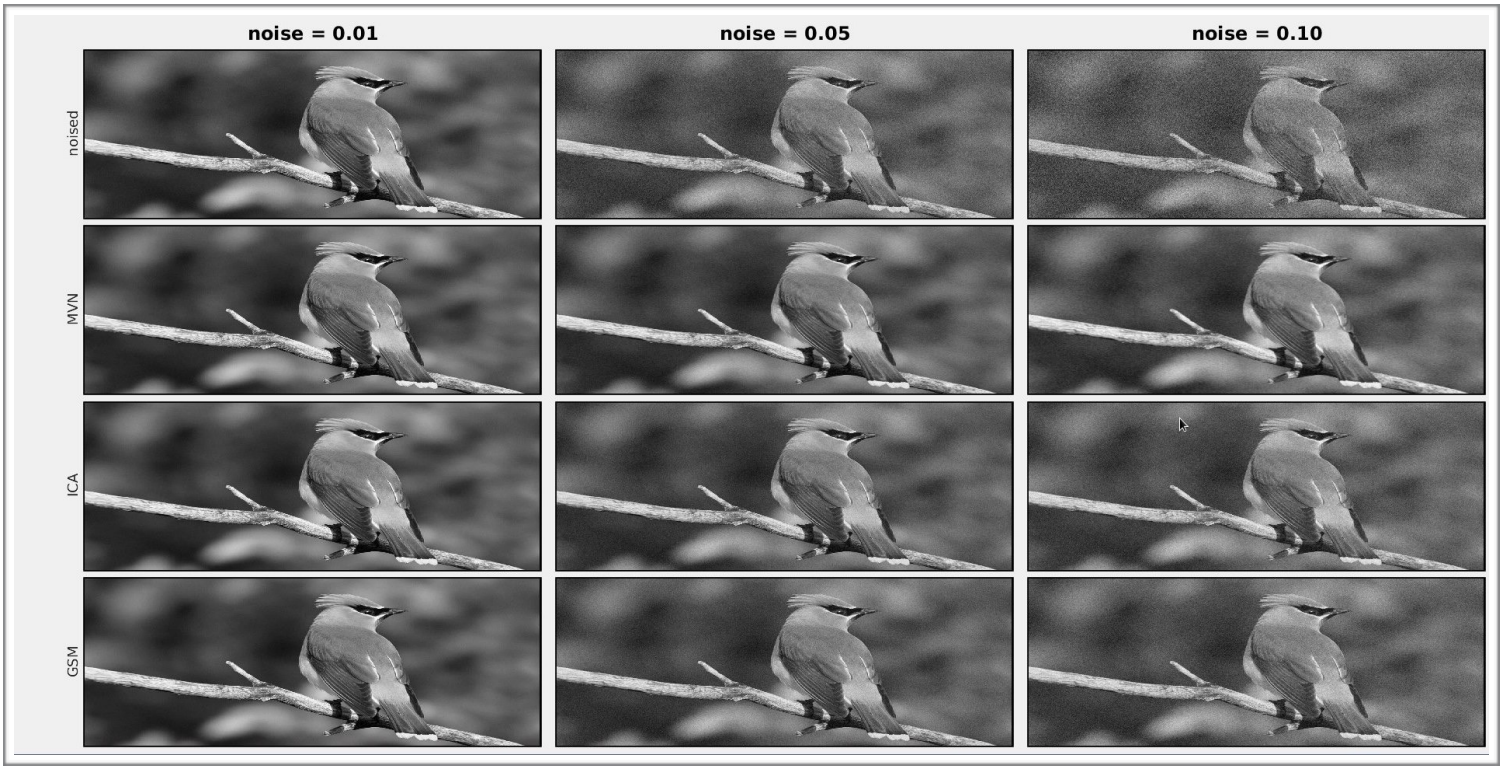


## Analysis

We noticed that in terms of pSNR, the MVN model is constantly better, while ICA and GSM approaches achieve better results in terms of log-likelihood.
The fact that MVN is a "simpler" mathematical model, and that it achieves a general approximation of the patches that appears visually more blurry, can explain the better results in terms of pSNR, since the pSNR value is dominated by the euclidean distance between the denoised patch and the original. Thus, a simple model that takes local means for each pixel (or something similar to that), may achieve good results in terms of euclidean distance, since the local mean minimizes this distance. On the other hand, a complex model that reconstructs a complicated pattern (rather than simply blurring the picture), might actually get worse results in per-pixel distance, since the pattern might not match the original pixels exactly, but still look more sharp. Indeed, we can see that the log-likelihood of the ICA and GSM are better, since they provide a better approximation of the original probability distribution, and thus provide a better denoising methods (as is visually apparent). To summarize the above, we believe that the ICA and GSM models provide better denoising, with GSM achieving slightly better qualitative results, while MVN results in a rather blurry image.

We also noticed that increasing the number of Gaussians (K) in the GSM and ICA models does not improve the model when K is large enough. This could be explained by the fact the having many Gaussians either results in a some Gaussians being very similar (and thus, having a similar denoising effect as simply selecting one such Gaussian), or some "extreme" Gaussians with almost zero mixture probability that have no effect on the denoising result.

## Denoising example for complete image:



# 2 ex1/GMM denoise.m

```
function [xhat] = GMM_denoise(y, gmm, noise)
% Denoises every column in y, assuming a gaussian mixture model and white
% noise.
%
% The model assumes that y = x + noise where x is generated from a GMM.
%
% Arguments
%  y - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  gmm - The mixture model, with 4 fields:
%          means - A KxD matrix where K is the number of components in
%                  mixture and D is the dimension of the data.
%          covs - A DxDxK array whose every page is a covariance matrix of
%                 the corresponding component.
%          mix - A Kx1 vector with mixing proportions.
%  noise - the std of the noise in y.
%

% ==================================================================
% This is an optional file - use if if you want to implement all denoising
% code in one place...
% ==================================================================

%XXX your denoising code here
[D, M] = size(y);
K = size(gmm.mix,2);
xhat = zeros(size(y));
for i=1:D
    HProbability = ICA_get_HProbability(gmm.mix(i,:), gmm.covs(i,:), y(i,:));
    denoise_factor = (1+((noise^2)./gmm.covs(i,:)));
    xhat(i,:) = sum((repmat(y(i,:),K,1)./repmat(denoise_factor',1,M)) .* HProbability');
end
```

# 3 ex1/GMM loglikelihood.m

```
function [ll] = GMM_loglikelihood(X, theta)
% Calculate the log likelihood of X, given a mixture model.
%
% The model assumes each column of x is independently generated by a
% mixture model with parameters theta.
%
% Arguments
%  X - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  theta - A struct with fields:
%          means - A KxD matrix where K is the number of components in
%                  mixture and D is the dimension of the data.
%          covs - A DxDxK array whose every page is a covariance matrix of
%                 the corresponding component.
%          mix - A Kx1 vector with mixing proportions.
%

% ========================================================================
% This is an optional file
% ========================================================================

[~,M] = size(X);
[K,~] = size(theta.mix);
log_k_prob = zeros(M,K);
for k=1:K
    if length(size(theta.covs)) > 2
        log_mvn = log_mvnpdf(X', theta.means(k,:), theta.covs(:,:,k));
    else
        log_mvn = log_mvnpdf(X', theta.means(k,:), theta.covs(k));
    end
    log_k_prob(:,k) = log_mvn + log(theta.mix(k));
end

ll = sum(logsum(log_k_prob,2),1);
```

# 4 ex1/GSM denoise.m

```
function [xhat] = GSM_denoise(y, gsm, noise)
% Denoises every column in y, assuming a GSM model and white noise.
%
% The model assumes that y = x + noise where x is generated by a mixture of
% 0-mean gaussian components sharing the same covariance up to a
% scaling factor
%
% Arguments
%  y - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  gsm - a struct with 3 fields:
%           mix - Mixture proportions.
%           covs - A DxDxK array, whose every page is a scaled covariance
%                  matrix according to scaling parameters.
%           means - K 0-means.
%  noise - the std of the noise in y.
%
[D, M] = size(y);
K = size(gsm.mix,1);


HProbability = get_HProbability(gsm.mix', gsm.covs, y);
xhat = zeros(D,M);
for k=1:K
   xhat = xhat+(inv((1/(noise^2)*eye(size(gsm.covs(:,:,k)))+inv(gsm.covs(:,:,k))))*(y/noise^2)).*repmat(HProbability(:,k)',D
end
```

# 5 ex1/GSM loglikelihood.m

```
function [ll] = GSM_loglikelihood(X, model)
% Calculate the log likelihood of X, given a GSM model.
%
% The model assumes that y = x + noise where x is generated by a mixture of
% 0-mean gaussian components sharing the same covariance up to a scaling
% factor.
%
% Arguments
%  X - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  model - a struct with 3 fields:
%           mix - Mixture proportions.
%           covs - A DxDxK array, whose every page is a scaled covariance
%                  matrix according to scaling parameters.
%           means - K 0-means.
%

ll = GMM_loglikelihood(X, model);
```

# 6 ex1/ICA denoise.m

```
function [xhat] = ICA_denoise(y, ica, noise)
% Denoises every column in y, assuming an ICA model and white noise.
%
% The model assumes that y = x + noise where x is generated by a ICA
% 0-mean mixture model.
%
% Arguments
%  y - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  ica - A struct with fields:
%           P - mixing matrix of sources (P: D ind. sources -> D signals)
%           vars - a DxK matrix whose (d,k) element correponsds to the
%                  variance of the k'th component in dimension d.
%           mix - a DxK matrix whose (d,k) element correponsds to the
%                 mixing weight of the k'th component in dimension d.
%  noise - the std of the noise in y.
%

[D, M] = size(y);
K = size(ica.mix,2);
y_s = inv(ica.P)*y;
y_s_denoise = zeros(size(y_s));

% iterate on each dimension to denoise individually
for i=1:D
    HProbability = get_HProbability(ica.mix(i,:), ica.vars(i,:), y_s(i,:));
    for k=1:K
        y_s_denoise(i,:) = y_s_denoise(i,:)+(inv((1/(noise^2)*eye(size(ica.vars(i,k)))+inv(ica.vars(i,k))))*(y_s(i,:)/noise^
    end
end

xhat = ica.P*y_s_denoise;
```

# 7 ex1/ICA loglikelihood.m

```
function [ll] = ICA_loglikelihood(X, model)
% Calculate the log likelihood of X, given an ICA model.
%
% The model assumes each column of x is independently generated by a
% mixture model with parameters theta.
%
% Arguments
%  X - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  model - A struct with fields:
%           P - mixing matrix of sources (P: D ind. sources -> D signals)
%           vars - a DxK matrix whose (d,k) element correponsds to the
%                  variance of the k'th component in dimension d.
%           mix - a DxK matrix whose (d,k) element correponsds to the
%                 mixing weight of the k'th component in dimension d.
%

[D, M] = size(X);
K = size(model.mix,2);
y_s = inv(model.P)*X;


ll = 0;
for i=1:D
    theta.covs = model.vars(i,:);
    theta.mix = model.mix(i,:)';
    theta.means = zeros(K,1);
    ll = ll + GMM_loglikelihood(y_s(i,:), theta);
end
```

# 8 ex1/MVN denoise.m

```
function [xhat] = MVN_denoise(y, mvn, noise)
% Denoises every column in y, assuming an MVN model and white noise.
%
% The model assumes that y = x + noise where x is generated by a single
% 0-mean multi-variate normal distribution.
%
% Arguments
%  y - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  mvn - A struct with field:
%           cov - A DxD covariane matrix.
%  noise - the std of the noise in y.
%

xhat = inv((1/(noise^2)*eye(size(mvn.cov))+inv(mvn.cov)))*(y/noise^2);
```

# 9 ex1/MVN loglikelihood.m

```
function [ll] = MVN_loglikelihood(X, model)
% Calculate the log likelihood of X, given an ICA model.
%
% The model assumes each column of x is independently generated by single
% 0-mean gaussian.
%
% Arguments
%  X - A DxM matrix, whose every column corresponds to a patch in D
%      dimensions (typically D=64).
%  model - A struct with field:
%           cov - A DxD covariane matrix.
%

ll = sum(log_mvnpdf(X', [], model.cov));
```

# 10 ex1/denoise.m

```
function [xhat] = denoise(y, model, noisestd, psize)
% denoise image y.
%
% This function splits y to subims in order to keep memory requirements
% down. To tweak subims size see MAX_SIZE variable below.
%
% Arguments:
%   y - a noisy gray-scale image.
%   model - a prior model with a "denoise" function handle field.
%   psize - the size of a patch use in model, if scalar interpreted as
%           square.
%   noisestd - the std of the noise.
%
if isscalar(psize) psize = [psize, psize]; end;

MAX_SIZE = [300,300];
xhat = nan(size(y));

%split to smaller images for memory considerations
[subims, rects] = imsplit(y, MAX_SIZE, psize);
[K,L] = size(subims);
pind = sub2ind(psize, ceil(psize(1)/2), ceil(psize(2)/2)); %middle index in patch
for k = 1:K
    for l = 1:L
        sub = subims{k,l};
        rect = rects{k,l};
        noisy_patches = im2col(sub, psize);
        if ~all(size(noisy_patches) > 0) continue, end
        innerh = rect(3)-psize(1)+1;
        innerw = rect(4)-psize(2)+1;
        if innerh > 0 && innerw > 0
            patches = model.denoise(noisy_patches, noisestd);
            cleaned = reshape(patches(pind,:),[innerh, innerw]);
            fr_i = rect(1) + ceil(psize(1)/2) - 1;
            to_i = rect(1) + innerh + 2;
            fr_j = rect(2) + ceil(psize(2)/2) - 1;
            to_j = rect(2) + innerw + 2;
            xhat(fr_i:to_i,fr_j:to_j) = cleaned;
        end
    end
end
end
```

# 11 ex1/get HProbability.m

```
function [H_probability_matrix] = get_HProbability(mix, cov, X)
% caclulate P(x_i|h=k)*P(h=k)/(sum_for_j-1:K(P(h=j,x_i))
M= length(X);
K = length(mix);
PDF_matrix = zeros(M,K);
for j=1:K
    if length(size(cov)) > 2
        PDF_matrix(:,j) = log_mvnpdf(X',0,cov(:,:,j));
    else
        PDF_matrix(:,j) = log_mvnpdf(X',0,cov(j));
end
numerator = PDF_matrix+log(repmat(mix,M,1));
denominator = logsum(PDF_matrix + log(repmat(mix,M,1)),2);
H_probability_matrix = exp(numerator - repmat(denominator,1,K));
if sum(sum(isnan(H_probability_matrix))) > 0
    keyboard
end
end
```

# 12 ex1/imsplit.m

```
function [subx, rects] = imsplit(X, max_size, overlap)
% split an image to subimages.

% Arguments:
%   X - the image.
%   max_size - either a scalar which is interpreted as a square, or a pair
%              [max_rows, max_cols]. This determines the maximal size
%              allowed for a resulting sub-image. default = half the image
%              size.
%   overlap - either a scalar which is interpreted as a square, or a pair
%             [row_overlap, col_overlap]. This determines the overlap in
%             rows and columns between subimages. default = 0.
% Returns:
%   subx - a cell array holding sub-images of X, in their order, so that
%          they cover X completely.
%   rects - a corresponding collection of tuples: (i, j, h, w) such that:
%           subx{k,l} = X(i:i+h-1,j:j+w-1)
%           rects is given as a 1x4 vector.
%

if ~exist('max_size','var') || isempty(max_size) max_size = ceil(size(X)/2); end;
if ~exist('overlap','var') || isempty(overlap) overlap = 0; end;
if isscalar(max_size) max_size = [max_size, max_size]; end;
if isscalar(overlap) overlap = [overlap, overlap]; end;

[M,N] = size(X);

sub_max_h = ceil(M / (max_size(1)-overlap(1)) );
sub_max_w = ceil(N / (max_size(2)-overlap(2)) );

Is = (1:floor(M/sub_max_h):M);
Js = (1:floor(N/sub_max_w):N);
rects = cell(length(Is),length(Js));
subx = cell(length(Is),length(Js));

for i_ind = 1:length(Is)
    i = Is(i_ind);
    h = min(i + max_size(1) - 1, M) - i + 1;
    for j_ind = 1:length(Js)
        j = Js(j_ind);
        w = min(j + max_size(2) - 1, N) - j + 1;
        subx{i_ind,j_ind} = X(i:i+h-1, j:j+w-1);
        rects{i_ind,j_ind} = [i,j,h,w];
    end
end
```

# 13 ex1/learn GMM.m

```
function [theta, LL] = learn_GMM(X, K, params0, options)
% Learn parameters for a gaussian mixture model via EM.
%
% Arguments:
%   X - Data, a DxM data matrix, where D is the dimension, and M is the
%       number of samples.
%   K - Number of components in mixture.
%   params0 - An optional struct with intialization parameters. Has 3
%             optional fields:
%               means - a KxD matrix whose every row corresponds to a
%                       component mean.
%               covs - A DxDxK array, whose every page is a component
%                      covariance matrix.
%               mix - A Kx1 mixture vector (should sum to 1).
%             If not given, a random starting point is generated.
%   options - Algorithm options struct, with fields:
%             learn - A struct of booleans, denoting which parameters
%                     should be learned: learn.means, learn.covs and
%                     learn.mix. The default is that given parameters
%                     (in params0) are not learned.
%             max_iter - maximum #iterations. Default = 100.
%             thresh - if previous_LL * thresh > current_LL,
%                      algorithm halts. default = 1.01.
%             verbosity - either 'none', 'iter' or 'plot'. default 'none'.
% Returns:
%   params - A struct with learned parameters (fields):
%               means - a KxD matrix whose every row corresponds to a
%                       component mean.
%               covs - A DxDxK array, whose every page is a component
%                      covariance matrix.
%               mix - A Kx1 mixture vector.
%   LL - log likelihood history
%
% ========================================================================
% This is an optional file - use it if you want to implement a single EM
% algorithm
% ========================================================================
%
EPS = 1e-10;
if ~exist('params0', 'var') params0 = struct(); end
[theta, default_learn] = get_params0(X, K, params0);

if ~exist('options', 'var') options = struct(); end
options = organize_options(options, default_learn);

[D,M] = size(X);

LL = [];

% iterate until convergence
for iter=1:options.max_iter

    % update posterior probability
    H_probability_matrix = get_HProbability(theta.mix',theta.covs,X);

    % update cov
    if options.learn.covs
        log_cov = log(X.*X);
        theta.covs(1,1,:) = exp(logsum(log(H_probability_matrix)+repmat(log_cov',1,K)))./sum(H_probability_matrix);
```

```
    end

    % update mix
    if options.learn.mix
        theta.mix = (sum(H_probability_matrix)/M)';
    end

    % update log-likelihood
    LL(iter) = GMM_loglikelihood(X, theta);


    % break if converged
    if (iter > 1) && (LL(iter-1)*options.threshold > LL(iter))
        break
    end

end

if strcmp(options.verbosity,'plot')
    plot(LL);
    pause(0.1)
end
end


function [params0, default_learn] = get_params0(X, K, params0)
% organizes the params0 struct and output the starting point of the
% algorithm - "params0".
default_learn.mix = false;
default_learn.means = false;
default_learn.covs = false;

[D,M] = size(X);

if ~isfield(params0, 'means')
    default_learn.means = true;
    params0.means = X(:,randi(M, [1,K]))';
    params0.means = params0.means + nanstd(X(:))*randn(size(params0.means));
end

if ~isfield(params0, 'covs')
    default_learn.covs = true;
    params0.covs = nan(D,D,K);
    for k = 1:K
        params0.covs(:,:,k) = nancov(X(:,randi(M, [1,100]))');
    end
end

if ~isfield(params0, 'mix')
    default_learn.mix = true;
    params0.mix = rand(K,1);
    params0.mix = params0.mix / sum(params0.mix);
end

end

function [options] = organize_options(options, default_learn)
%organize the options.
if ~isfield(options, 'threshold') options.threshold = 1.01; end
if ~isfield(options, 'max_iter') options.max_iter = 100; end
if ~isfield(options, 'verbosity') options.verbosity = 'none'; end
if ~isfield(options, 'learn') options.learn = default_learn;
else
    if ~isfield(options.learn, 'means') options.learn.means = default_learn.means; end;
    if ~isfield(options.learn, 'covs') options.learn.covs = default_learn.covs; end;
    if ~isfield(options.learn, 'mix') options.learn.mix = default_learn.mix; end;
end
end
```

# 14 ex1/learn GSM.m

```
function [model] = learn_GSM(X, K, options)
% Learn parameters for a gaussian scaling mixture model for X via EM
%
% GSM components share the variance, up to a scaling factor, so we only
% need to learn scaling factors c_1.. c_K and mixture proportions
% alpha_1..alpha_K.
%
% Arguments:
%   X - Data, a DxM data matrix, where D is the dimension, and M is the
%       number of samples.
%   K - Number of components in mixture.
%   options - options for learn_GMM (optional).
% Returns:
%   model - a struct with 3 fields:
%           mix - Mixture proportions.
%           covs - A DxDxK array, whose every page is a scaled covariance
%                  matrix according to scaling parameters.
%           means - K 0-means.
%

[D,M] = size(X);
Z = cov(X');
c = rand(1,1,K)*10;
params.covs = repmat(Z,[1,1,K]) .* repmat(c,[D,D,1]);
params.means = zeros(K,1);

options.learn.covs = false;
options.learn.means = false;
options.learn.mix = true;

theta = learn_GMM(X, K, params, options);
model.mix = theta.mix;
model.covs = theta.covs;
model.means = theta.means;
```

# 15 ex1/learn ICA.m

```
function model = learn_ICA(X, K, options)
% Learn parameters for a complete invertible ICA model.
%
% We learn a matrix P such that X = P*S, where S are D independent sources
% And for each of the D coordinates we learn a mixture of K (univariate)
% 0-mean gaussians via EM.
%
% Arguments:
%   X - Data, a DxM data matrix, where D is the dimension, and M is the
%       number of samples.
%   K - Number of components in a mixture.
%   options - options for learn_GMM (optional).
% Returns:
%   model - A struct with 3 fields:
%           P - mixing matrix of sources (P: D ind. sources -> D signals)
%           vars - a DxK matrix whose (d,k) element correponsds to the
%                  variance of the k'th component in dimension d.
%           mix - a DxK matrix whose (d,k) element correponsds to the
%                 mixing weight of the k'th component in dimension d.
%

% get the P matrix, by multiplying the eigenvectors of the covariance with the squared eigenvalue matrix
[D,M] = size(X);

% PCA
[P, Lambda] = eig(cov(X'));
model.P = P;
S = inv(model.P)*X;

model.mix = zeros(D,K);
model.vars = zeros(D,K);

% learn each dimension individually
for d=1:D
    params.means = zeros(K,1);
    theta = learn_GMM(S(d,:), K, params);
    model.mix(d,:) = theta.mix;
    model.vars(d,:) = theta.covs;
end


end
```

# 16 ex1/learn MVN.m

```
function [model] = learn_MVN(X)
% Learn parameters for a 0-mean multivariate normal model for X.
%
% Arguments:
%   X - Data, a DxM data matrix, where D is the dimension, and M is the
%       number of samples.
%   K - Number of components in mixture.
%   options - options for learn_GMM (optional).
% Returns:
%   model - a struct with 3 fields:
%           cov - DxD covariance matrix.
%
model.cov = cov(X');
```

# 17 ex1/log mvnpdf.m

```
function log_pr = log_mvnpdf(X, Mu, Sigma)
%MVNPDF Multivariate normal probability density function (pdf).
%   Y = MVNPDF(X) returns the probability density of the multivariate normal
%   distribution with zero mean and identity covariance matrix, evaluated at
%   each row of X.  Rows of the N-by-D matrix X correspond to observations or
%   points, and columns correspond to variables or coordinates.  Y is an
%   N-by-1 vector.
%
%   Y = MVNPDF(X,MU) returns the density of the multivariate normal
%   distribution with mean MU and identity covariance matrix, evaluated
%   at each row of X.  MU is a 1-by-D vector, or an N-by-D matrix, in which
%   case the density is evaluated for each row of X with the corresponding
%   row of MU.  MU can also be a scalar value, which MVNPDF replicates to
%   match the size of X.
%
%   Y = MVNPDF(X,MU,SIGMA) returns the density of the multivariate normal
%   distribution with mean MU and covariance SIGMA, evaluated at each row
%   of X.  SIGMA is a D-by-D matrix, or an D-by-D-by-N array, in which case
%   the density is evaluated for each row of X with the corresponding page
%   of SIGMA, i.e., MVNPDF computes Y(I) using X(I,:) and SIGMA(:,:,I).
%   If the covariance matrix is diagonal, containing variances along the
%   diagonal and zero covariances off the diagonal, SIGMA may also be
%   specified as a 1-by-D matrix or a 1-by-D-by-N array, containing
%   just the diagonal. Pass in the empty matrix for MU to use its default
%   value when you want to only specify SIGMA.
%
%   If X is a 1-by-D vector, MVNPDF replicates it to match the leading
%   dimension of MU or the trailing dimension of SIGMA.
%
%   Example:
%
%      mu = [1 -1]; Sigma = [.9 .4; .4 .3];
%      [X1,X2] = meshgrid(linspace(-1,3,25)', linspace(-3,1,25)');
%      X = [X1(:) X2(:)];
%      p = mvnpdf(X, mu, Sigma);
%      surf(X1,X2,reshape(p,25,25));
%
%   See also MVTPDF, MVNCDF, MVNRND, NORMPDF.

%   Copyright 1993-2008 The MathWorks, Inc.
%   $Revision: 1.1.8.2 $  $Date: 2010/10/08 17:25:11 $

if nargin<1
    error(message('stats:mvnpdf:TooFewInputs'));
elseif ndims(X)~=2
    error(message('stats:mvnpdf:InvalidData'));
end

% Get size of data.  Column vectors provisionally interpreted as multiple scalar data.
[n,d] = size(X);
if d<1
    error(message('stats:mvnpdf:TooFewDimensions'));
end

% Assume zero mean, data are already centered
if nargin < 2 || isempty(Mu)
    X0 = X;

% Get scalar mean, and use it to center data
```

```
elseif numel(Mu) == 1
    X0 = X - Mu;

% Get vector mean, and use it to center data
elseif ndims(Mu) == 2
    [n2,d2] = size(Mu);
    if d2 ~= d % has to have same number of coords as X
        error('stats:mvnpdf:InputSizeMismatch',...
              'X and MU must have the same number of columns.');
    elseif n2 == n % lengths match
        X0 = X - Mu;
    elseif n2 == 1 % mean is a single row, rep it out to match data
        X0 = bsxfun(@minus,X,Mu);
    elseif n == 1 % data is a single row, rep it out to match mean
        n = n2;
        X0 = bsxfun(@minus,X,Mu);
    else % sizes don't match
        error('stats:mvnpdf:InputSizeMismatch',...
              ['X or MU must be a row vector, or X and MU must have the '...
              'same number of rows.']);
    end

else
    error(message('stats:mvnpdf:BadMu'));
end

% Assume identity covariance, data are already standardized
if nargin < 3 || isempty(Sigma)
    % Special case: if Sigma isn't supplied, then interpret X
    % and Mu as row vectors if they were both column vectors
    if (d == 1) && (numel(X) > 1)
        X0 = X0';
        d = size(X0,2);
    end
    xRinv = X0;
    logSqrtDetSigma = 0;

% Single covariance matrix
elseif ndims(Sigma) == 2
    sz = size(Sigma);
    if sz(1)==1 && sz(2)>1
        % Just the diagonal of Sigma has been passed in.
        sz(1) = sz(2);
        sigmaIsDiag = true;
    else
        sigmaIsDiag = false;
    end

    % Special case: if Sigma is supplied, then use it to try to interpret
    % X and Mu as row vectors if they were both column vectors.
    if (d == 1) && (numel(X) > 1) && (sz(1) == n)
        X0 = X0';
        d = size(X0,2);
    end

    %Check that sigma is the right size
    if sz(1) ~= sz(2)
        error('stats:mvnpdf:BadCovariance',...
              'SIGMA must be a square matrix or a vector.');
    elseif ~isequal(sz, [d d])
        error('stats:mvnpdf:InputSizeMismatch',...
              ['SIGMA must be a square matrix with size equal to the ' ...
              'number of columns in X, or a row vector with length '...
              'equal to the number of columns in X.']);
    else
        if sigmaIsDiag
            if any(Sigma<=0)
                error(message('stats:mvnpdf:BadDiagSigma'));
```

```
            end
            R = sqrt(Sigma);
            xRinv = bsxfun(@rdivide,X0,R);
            logSqrtDetSigma = sum(log(R));
        else
            % Make sure Sigma is a valid covariance matrix
            [R,err] = cholcov(Sigma,0);
            if err ~= 0
                error('stats:mvnpdf:BadCovariance', ...
                    'SIGMA must be symmetric and positive definite.');
            end
            % Create array of standardized data, and compute log(sqrt(det(Sigma)))
            xRinv = X0 / R;
            logSqrtDetSigma = sum(log(diag(R)));
        end
    end

% Multiple covariance matrices
elseif ndims(Sigma) == 3

    sz = size(Sigma);
    if sz(1)==1 && sz(2)>1
        % Just the diagonal of Sigma has been passed in.
        sz(1) = sz(2);
        Sigma = reshape(Sigma,sz(2),sz(3))';
        sigmaIsDiag = true;
    else
        sigmaIsDiag = false;
    end

    % Special case: if Sigma is supplied, then use it to try to interpret
    % X and Mu as row vectors if they were both column vectors.
    if (d == 1) && (numel(X) > 1) && (sz(1) == n)
        X0 = X0';
        [n,d] = size(X0);
    end

    % Data and mean are a single row, rep them out to match covariance
    if n == 1 % already know size(Sigma,3) > 1
        n = sz(3);
        X0 = repmat(X0,n,1); % rep centered data out to match cov
    end

    % Make sure Sigma is the right size
    if sz(1) ~= sz(2)
        error('stats:mvnpdf:BadCovariance',...
              'Each page of SIGMA must be a square matrix or a row vector.');
    elseif (sz(1) ~= d) || (sz(2) ~= d) % Sigma is a stack of dxd matrices
        error('stats:mvnpdf:InputSizeMismatch',...
              ['Each page of SIGMA must be a square matrix with size equal '...
              'to the number of columns in X, or a row vector with length '...
              'the same as the number of columns in X.']);
    elseif sz(3) ~= n
        error('stats:mvnpdf:InputSizeMismatch',...
              'SIGMA must have one page for each row of X.');
    else
        if sigmaIsDiag
            if any(any(Sigma<=0))
                error(message('stats:mvnpdf:BadDiagSigma'));
            end
            R = sqrt(Sigma);
            xRinv = X0./R;
            logSqrtDetSigma = sum(log(R),2);
        else
            % Create array of standardized data, and vector of log(sqrt(det(Sigma)))
            xRinv = zeros(n,d,superiorfloat(X0,Sigma));
            logSqrtDetSigma = zeros(n,1,class(Sigma));
            for i = 1:n
```

```
                % Make sure Sigma is a valid covariance matrix
                [R,err] = cholcov(Sigma(:,:,i),0);
                if err ~= 0
                    error('stats:mvnpdf:BadCovariance',...
                        'Each page of SIGMA must be symmetric and positive definite.');
                end
                xRinv(i,:) = X0(i,:) / R;
                logSqrtDetSigma(i) = sum(log(diag(R)));
            end
        end
    end

elseif ndims(Sigma) > 3
    error('stats:mvnpdf:BadCovariance',...
          'SIGMA must be a vector, a matrix, or a 3 dimensional array.');
end

% The quadratic form is the inner products of the standardized data
quadform = sum(xRinv.^2, 2);

log_pr = -0.5*quadform - logSqrtDetSigma - d*log(2*pi)/2;
```

# 18 ex1/lognormat.m

```
function N = lognormat(A, dim)
%LOGNORMAT normalize A along dim, in log space.
% Arguments:
%   A - matrix to normalize
%   dim - dimension along which to normalize. default is 2
%         (column)

if ~exist('dim','var') dim = 2; end;

rep_dims = ones(1,length(size(A))); rep_dims(dim) = size(A, dim);
N =  A - repmat(logsum(A,dim), rep_dims);
end
```

# 19 ex1/logsum.m

```
function S = logsum(X, dim)
%LOGSUM executes: S = log(sum(exp(X)), along dim in a precision-aware
% manner.
%
%   Arguments:
%       X - a matrix in log space.
%       dim - the dimension along which summation is performed,
%             defaults to 1, i.e. rows.
if ~exist('dim','var') || isempty(dim) dim = 1; end
c = size(X, dim); rep_dims = ones(1,length(size(X))); rep_dims(dim) = c;
m = max(X, [], dim);
M = repmat(m, rep_dims);
S = squeeze(m + log(sum(exp(X-M),dim)));
S(isnan(S)) = -inf; %only reason for a NaN is -inf - -inf
end
```

# 20 ex1/main.m

```
function main(learningMethod, patchSize, N, noise)
load 'ims.mat';
ims.train = standardize_ims(ims.train);
ims.test = standardize_ims(ims.test);
ps = sample_patches(ims.train, patchSize,N,false);
K = 5;
i = 1;
if learningMethod(1) == 1
    model{i} = learn_MVN(ps);
    model{i}.name = 'MVN';
    model{i}.loglikelihood = @(x)MVN_loglikelihood(x,model{i});
    model{i}.denoise = @(y, noise)MVN_denoise(y, model{i}, noise);
    i = i + 1;
end
if learningMethod(2) == 1
    model{i} = learn_ICA(ps,K);
    model{i}.name = 'ICA';
    model{i}.loglikelihood = @(x)ICA_loglikelihood(x,model{i});
    model{i}.denoise = @(y, noise)ICA_denoise(y, model{i}, noise);
    i = i + 1;
end
if learningMethod(3) == 1
    model{i} = learn_GSM(ps, K);
    model{i}.name = 'GSM';
    model{i}.loglikelihood = @(x)GSM_loglikelihood(x,model{i});
    model{i}.denoise = @(y, noise)GSM_denoise(y, model{i}, noise);
end

[psnr, ll] = test_denoising(ims.test(2:3:8),model, noise, [100,160], true, patchSize);
disp(ll)
disp(psnr)

end
```

# 21 ex1/normat.m

```
function [N, S] = normat(A, dim)
%NORMATnormalize A along dim.
% Arguments:
%   A - matrix to normalize
%   dim - dimension along which to normalize. default is 2
%         (column)
%
% Returns:
%   N - the normalized (sums to 1 along dim) matrix.
%   S - the sums along dims.
%

if ~exist('dim','var') dim = 2; end;

rep_dims = ones(1,length(size(A))); rep_dims(dim) = size(A, dim);
S = nansum(A,dim);
N =  A ./ repmat(S, rep_dims);
end
```

# 22 ex1/sample patches.m

```
function [ps] = sample_patches(ims, psize, N, rmmean)
% sample N psized patches from ims uniformly
%
% Arguments:
%   ims - a cell array containing images.
%   psize - the size of the patch, if scalar assumed to be square
%           (default = 8)
%   N - number of total patches to sample. default = 1e4.
%   rmmean - whether mean should be removed. default = true
%
% Returns:
%   ps - N psized patches.
%
if ~exist('psize','var') || isempty(psize) psize = 8; end;
if ~exist('N','var') || isempty(N) N = 1e3; end;
if ~exist('rmmean','var') || isempty(rmmean) rmmean = true; end;
if isscalar(psize) psize = [psize, psize]; end;

P = numel(nan(psize));
nim = length(ims);
sizes = cellfun(@(x)numel(x),ims);

if rmmean
    mu = sum(cellfun(@(x)sum(x(:)),ims))./sum(sizes);
end
to_sample = ceil(sizes .* N / sum(sizes)); %how many patches to sample from each image
while sum(to_sample) > N
    i = randi(nim,1);
    to_sample(i) = to_sample(i) - 1;
end

ps = nan(P, N);
frm = [0,cumsum(to_sample(1:end-1)),N];
for i = 1:nim
    %random indices in current image
    idx = nan(P,to_sample(i));
    row_idx = randi(size(ims{i},1)-psize(1),[1,to_sample(i)]);
    col_idx = randi(size(ims{i},2)-psize(2),[1,to_sample(i)]);
    idx(1,:) = sub2ind(size(ims{i}),row_idx,col_idx)';
    for j = 2:psize(1)
        idx(j,:) = idx(j-1,:) + 1;
    end
    for j = 2:psize(2)
        idx((j-1)*psize+1:psize(1)*j,:) = idx(1:psize(1),:) + j * size(ims{i},1);
    end

    %add to patches
    ps(:,frm(i)+1:frm(i+1)) = reshape(ims{i}(idx(:)), [P, to_sample(i)]);
end

if rmmean
    ps = ps - mu;
end
end
```

# 23 ex1/standardize ims.m

```
function [sims] = standardize_ims(ims)
% converts images to greyscale, rescals to 0-1 and removes the mean pixel
% value of the entire dataset.

sims = cellfun(@(x)double(rgb2gray(x))/255, ims, 'uniformoutput', false);
mu = sum(cellfun(@(x)sum(x(:)), sims)) / sum(cellfun(@(x)numel(x), sims));
sims = cellfun(@(x)x-mu, sims, 'uniformoutput', false);
```

# 24 ex1/test denoising.m

```
function [psnr, ll, dur] = test_denoising(test, models, noise_range,...
                                          to_plot, only_plot, psize)
% Arguments:
%  test - a cell array of images.
%  models - prior models. structs with fields:
%           * denoise - a function xhat <- y,noise
%           * loglikelihood - a function R <- x
%           * name - a string with the name of the model.
%  noise_range - a range of noise to be added to the picture.
%                default = [.001, .05, .1, .2].
%  to_plot - whether results should be plotted. default = [], i.e. don't.
%            If a pair [width, height] is given, it's interpreted as a
%            frame size to be plotted.
%  psize - patch size, either scalar (square ptch) or a pair. defaults to
%          8.
%  only_plot - if true, only the frames that are plotted are denoised
%              rather than complete images. useful for debugging.
% Returns:
%   psnr - a Ix(1+M)xN array with results where I=#images, M=#models,
%          N=#noise.
%   ll - a M lengthed vector of log likelihood of each model (estimated via
%        10^5).
%   dur - the duration it took to denoise the images (a Ix(1+M)xN array);
%
% Usage:
%
% %learn ica and prep model struct
% >> ica_model = learn_ICA(train_x, 10, options);
% >> ica_model.name = 'ICA';
% >> ica_model.loglikelihood = @(x)ICA_loglikelihood(x, ica_model);
% >> ica_model.denoise = @(y, noise)ICA_denoise(y, ica_model, noise);
%
% %learn gsm and prep model struct
% >> gsm_model = learn_GSM(train_x, 20, options);
% >> gsm_model.name = 'GSM';
% >> gsm_model.loglikelihood = @(x)GMM_loglikelihood(x,gsm_model);
% >> gsm_model.denoise = @(y, noise)GMM_denoise(y,gsm_model,noise);
%
% % denoise, evaluate and plot
% >> [psnr, ll] = test_denoising(ims.test(2:3:8),{mvn_model, ica_model, gsm_model});
% ...
% >> bar(ll') % yay!

pSNR = @(x,y)-10*log10(nanmean((x(:)-y(:)).^2));

if ~exist('noise_range','var') || isempty(noise_range)
    noise_range = [.001, .05, .1];
end;
if ~exist('psize','var') || isempty(psize) psize = 8; end;
if ~exist('to_plot','var') || isempty(to_plot) to_plot = [100,160]; end;
if ~exist('only_plot','var') || isempty(only_plot) only_plot = true; end;

if isscalar(psize) psize = [psize, psize]; end;

%adding a model that does nothing to the mix
nomodel.denoise = @(y,noisestd) y;
nomodel.loglikelihood = @(x)0;
nomodel.name = 'noised';
models = {nomodel, models{:}};
```

```
if only_plot
    %keep only middle frame of images
    midframe = @(x) x((size(x,1) - to_plot(1))/2 - 1 : (size(x,1) + to_plot(1))/2 + 1,...
                      (size(x,2) - to_plot(2))/2 - 1: (size(x,2) + to_plot(2))/2 + 1);
    test = cellfun(midframe, test, 'uniformoutput', false);
end


I = length(test);
M = length(models);
S = length(noise_range);

psnr = nan(I,M,S);
dur = nan(I,M,S);
ll = nan(M,1);

if ~isempty(to_plot)
    pR = M;
    pC = S;
    ahs = nan(I,pR,pC); %axes handles for all figures
    for i = 1:I
        figure;
        clf;
        colormap(gray);
        ahs(i,:,:) = reshape(tight_subplot(pR, pC, [.01,.01],...
                                           [.01, .05], [.05, .01]), [S, M])';
    end
end

tst_ps = sample_patches(test, psize, 1e5);
for i = 1:I %images
    x = test{i};
    for si = 1:S %noise
        noise = noise_range(si);
        y = x + noise * randn(size(x));
        for mi = 1:M % models
            model = models{mi};
            fprintf('denoising image %i with %s (noise %2f).\n',...
                    i, model.name, noise)
            if isnan(ll(mi))
                ll(mi) = model.loglikelihood(tst_ps) ./...
                         (log(2) * numel(tst_ps)); %in bits/pixel
            end
            tic
            xhat = denoise(y, model, noise, psize);
            dur(i,mi,si) = toc;
            psnr(i,mi,si) = pSNR(x,xhat);
            if ~isempty(to_plot)
                lims = round((size(x) - to_plot)/2);
                lime = round((size(x) + to_plot)/2);
                axes(ahs(i,mi,si));
                imagesc(xhat(lims(1):lime(1),lims(2):lime(2)));
                set(gca, 'xtick',[],'ytick',[]);
                if mi == 1
                    title(sprintf('noise = %.2f', noise), 'fontsize', 18);
                end
                if si == 1
                    ylabel(model.name, 'fontsize', 14)
                end
            end
            pause(.1);
        end
    end
end
```

# 25 ex1/tight subplot.m

```
function ha = tight_subplot(Nh, Nw, gap, marg_h, marg_w)

% tight_subplot creates "subplot" axes with adjustable gaps and margins
%
% ha = tight_subplot(Nh, Nw, gap, marg_h, marg_w)
%
%   in:  Nh      number of axes in hight (vertical direction)
%        Nw      number of axes in width (horizontaldirection)
%        gap     gaps between the axes in normalized units (0...1)
%                   or [gap_h gap_w] for different gaps in height and width
%        marg_h  margins in height in normalized units (0...1)
%                   or [lower upper] for different lower and upper margins
%        marg_w  margins in width in normalized units (0...1)
%                   or [left right] for different left and right margins
%
%  out:  ha     array of handles of the axes objects
%                   starting from upper left corner, going row-wise as in
%                   going row-wise as in
%
%  Example: ha = tight_subplot(3,2,[.01 .03],[.1 .01],[.01 .01])
%           for ii = 1:6; axes(ha(ii)); plot(randn(10,ii)); end
%           set(ha(1:4),'XTickLabel',''); set(ha,'YTickLabel','')

% Pekka Kumpulainen 20.6.2010   @tut.fi
% Tampere University of Technology / Automation Science and Engineering


if nargin<3; gap = .02; end
if nargin<4 || isempty(marg_h); marg_h = .05; end
if nargin<5; marg_w = .05; end

if numel(gap)==1;
    gap = [gap gap];
end
if numel(marg_w)==1;
    marg_w = [marg_w marg_w];
end
if numel(marg_h)==1;
    marg_h = [marg_h marg_h];
end

axh = (1-sum(marg_h)-(Nh-1)*gap(1))/Nh;
axw = (1-sum(marg_w)-(Nw-1)*gap(2))/Nw;

py = 1-marg_h(2)-axh;

ha = zeros(Nh*Nw,1);
ii = 0;
for ih = 1:Nh
    px = marg_w(1);

    for ix = 1:Nw
        ii = ii+1;
        ha(ii) = axes('Units','normalized', ...
            'Position',[px py axw axh], ...
            'XTickLabel','', ...
            'YTickLabel','');
        px = px+axw+gap(2);
    end
```

```
    py = py-axh-gap(1);
end
```